National Defence
Research and Development Branch

Défense nationale
Bureau de recherche et développement

TECHNICAL COMMUNICATION 94/302
February 1994

C++ CLASSES FOR READING AND WRITING IN **OFFSRF** FORMAT

David Hally

Defence Research Establishment Atlantic

Centre de Recherches pour la Défense Atlantique

Canada

UNLIMITED DISTRIBUTION

National Defence
Research and Development Branch

Défense nationale
Bureau de recherche et développement

C++ CLASSES FOR READING AND WRITING IN **OFFSRF** FORMAT

David Hally

February 1994

Approved by R.T. Schmitke
Director / Technology Division

Distribution Approved by

Director / Technology Division

DREA TECHNICAL COMMUNICATION 94/302

Defence Research Establishment Atlantic

Centre de Recherches pour la Défense Atlantique

Canada

Abstract

The OFFSRF format defines a record structure for files to be used as input to computer programs. The original application was offset data for ship hulls, but the format itself is very general. It is most suitable for comparatively small sets of data which are often modified using ordinary text editors.

When OFFSRF was originally designed, several Fortran subroutines were provided to ease the use of the format for programmers. In this document the programmer support is extended to the language C++. C++ classes are described which can be used to read and write files in OFFSRF format.

Résumé

Le format OFFSRF définit une structure d'enregistrement pour les fichiers qui doivent servir aux entrées dans des logiciels. L'application initiale était celle des données de décalage pour les coques de navire, mais le format comme tel est très général. Il convient spécialement aux ensembles de données relativement petits et souvent modifiés à l'aide d'éditeurs de texte ordinaires.

Au moment de la conception d'OFFSRF, plusieurs sous-programmes en FORTRAN ont été fournis afin que les programmeurs puissent utiliser le format plus facilement. Ce document étend le soutient offert en englobant le langange C++, et il décrit des classes C++ qui peuvent s'utiliser pour la lecture et l'écriture avec des fichiers en format OFFSRF.

Contents

1 Introduction

The OFFSRF format for computer data files[1] was designed to provide a well-structured format for comparatively small files which are often modified using ordinary text editors. The original application was offset data for ship hulls, but the format itself is very general.

When OFFSRF was first proposed, several Fortran subroutines were provided to ease the use of the format for programmers. In this document the programmer support is extended to the language C++. C++ classes are described which can be used for reading and writing files in OFFSRF format.

To use the classes described here, you must include the header file `OFFSRF.h` in your application. Check with your system manager to see whether this file is installed on your system.

The OFFSRF classes make use of a representation of character strings defined by the `str` class. Its definition may be found in the file `str.h`. The member functions of the `str` class are described in Appendix A.

2 OFFSRF Input and Output Streams

`OFFSRF.h` defines two new I/O streams: one for input from an OFFSRF file, and one for output to an OFFSRF file. They are called `OFFSRF_ifstream` and `OFFSRF_ofstream` respectively and are specializations of the standard C++ I/O classes `ifstream` and `ofstream` normally defined in `fstream.h`.

2.1 Constructors

The constructors for the `OFFSRF_ifstream` and `OFFSRF_ofstream` classes both require a `char*` argument which is interpreted as the name of the file to be connected to the stream. There is a second, optional `int` argument which controls the stream's behaviour if it is unable to open the file: if the argument is one, an error message is written to `cout` and program execution is aborted; if it is zero, the execution continues but the value of the stream when interpreted as an `int` is zero. If the second argument is not given, its value defaults to one: i.e. a fatal error message is written. The declarations

```
OFFSRF_ifstream in("test.in");
```

```
OFFSRF_ifstream in("test.in",1);
```

and

```
OFFSRF_ifstream in("test.in",0);
if (!stream) open_file_err("test.in");
```

are all equivalent. The function `open_file_err()` writes the fatal error message.

2.2 File Paths

The HLLFLO programs[2], for which the OFFSRF format was originally developed, allow users to indicate a set of directories in which the HLLFLO programs should look for files. This is done by creating a file called `.filsub` in the user's home directory. Each line of this file contains a directory name. Whenever a HLLFLO program opens an existing file, it first looks for the file on the connected directory, then in each of the directories in `.filsub` until the file is found.

This file path mechanism is also implemented for the `OFFSRF_ifstream` class. When the stream is connected to a file, the directories in `$HOME/.filsub` will be searched until a file of the correct name is found.

2.3 Skipping of comments by `OFFSRF_ifstream`

An `OFFSRF_ifstream` overloads the `ws` manipulator so that OFFSRF comments are skipped as well as whitespace: i.e. any characters between an exclamation point and the end of line will be ignored. Moreover, the extractor `>>` is overloaded so that when variables of types `short`, `int`, `long`, `unsigned short`, `unsigned int`, `unsigned long`, `float`, or `double` are read, the whitespace and comments preceding them are *always* skipped. This is in conformance with the OFFSRF paradigm that the exact position of a number on a line should not be important. This is *not* true for `char` data for which whitespace may be significant.

For example, suppose that the file `test.in` contains the following data.

```
!This is a comment followed by a blank line

123
```

Then the program

```
#include "OFFSRF.h"
main()
{
  OFFSRF_ifstream stream("test.in");
  int i;
  stream >> i;
  cout << "i = " << i << endl;
}
```

will correctly output

```
i = 123
```

The OFFSRF comment line, the blank line, and the spaces before the number 123 are skipped.

When a string (either a variable of class `str` or a `char*`) is read by the inserter for an `ifstream`, the string is delimited by whitespace characters. An `OFFSRF_ifstream` allows the inclusion of non-end-of-line whitespace in the string. Use the manipulator `str_ws` to allow whitespace in the strings. Use the manipulator `no_str_ws` to forbid it. By default, whitespace will not be read into a string. Thus, when `str_ws` is in effect, a string is terminated by an

end-of-line, an end-of-file, a begin record character ('{'), or an end record character ('}'); when **no_str_ws** is in effect, a string is terminated by any whitespace, an end-of-file,the begin record character, or the end record character.

For example, if the file `test.in` contains the data

```
First line
Second line {with stuff in brackets }
```

then the code

```
main()
{
  OFFSRF_ifstream in1("test.in");
  str s1, s2, s3;
  in1 >> s1 >> s2 >> s3;
  cout << s1 << endl << s2 << endl << s3 << endl << endl;
  in1.close();

  OFFSRF_ifstream in2("test.in");
  char ch;
  in2 >> str_ws >> s1 >> s2;
  in2 >> ch; // gobble the begin record character
  in2 >> s3;
  cout << s1 << endl << s2 << endl << s3 << endl;
  in2.close();
}
```

will have the following output.

```
First
line
Second

First line
Second line
with stuff in brackets
```

2.4 OFFSRF_ofstream manipulators

An **OFFSRF_ofstream** provides several manipulators which ease the output of the start and end of an OFFSRF record. The manipulator **bgnrec(record_name)** begins a new record whose name is **record_name**. A **bgnrec** manipulator should always be matched by the manipulator **endrec**.

Use of **bgnrec** and **endrec** causes nested records to be indented automatically. This is implemented by overloading the manipulator **endl** so that it indents an appropriate number of spaces.

The name of an OFFSRF record can be followed by an optional colon to indicate that the remainder of the line contains significant data. The behaviour of **bgnrec** and **endrec** depends

on whether the colon format is used. Use of colon formatting is enabled with an optional second argument to **bgnrec**: if this argument is a non-zero integer, colon formatting will be used. The colon formatting is automatically disabled by the **endrec** and **endl** manipulators.

When colon formatting is enabled, **bgnrec** writes the name **record-name** preceded by a { and followed by a colon. It then increases the level of indenting by two spaces. **endrec** decrements the level of indenting by two spaces, writes a }, then disables the colon formatting.

If colon formatting is disabled, **bgnrec** does not write a colon after the record name, but it does start a new line, appropriately indented. **endrec** starts a new line after the level of indentation has been reduced, then writes the } followed by the record name.

For example

```
OFFSRF_ofstream stream("test.out");
stream << bgnrec("TEST RECORD") << 123.45 << endl
         << bgnrec("COMMENT",1) << "This is a comment." << endrec
       << endrec << endl
       << bgnrec("RECORD 2",1) << "  abcDEF" << endrec << endl
       << bgnrec("RECORD 3",1) << "  abcDEF" << endl << endrec << endl;
```

causes the output in the file **test.out**:

```
{TEST RECORD
  123.45
  {COMMENT:This is a comment.}
}TEST RECORD
{RECORD 2:  abcDEF}
{RECORD 3:  abcDEF

}RECORD 3
```

Notice the difference between **RECORD 2** and **RECORD 3**. In the former, **endl** was not used prior to **endrec**, so that the colon formatting was enabled when **endrec** was called. In the latter, the **endl** manipulator had turned off the colon formatting before **endrec** was called.

2.5 The connected file

OFFSRF streams retain the name of the file to which they are connected. The name of the file may be obtained using the member function **file()** which returns a **str**. Thus,

```
OFFSRF_ifstream stream("test.in");
cout << "The connected file is " << stream.file() << endl;
```

will cause the output

```
The connected file is test.in
```

2.6 Verbose and terse modes

The OFFSRF streams can be used in either verbose mode or terse mode. When used in verbose mode, extra comments will be written to **cout** describing the state of the input or output. When in terse mode these optional comments will be suppressed.

An OFFSRF stream can be set to verbose mode using the manipulator **verbose**; it can be set to terse mode using the manipulator **terse**. The current mode of an OFFSRF stream may be obtained using the **int** function **is_verbose()** which returns zero if the stream is not in verbose mode, non-zero otherwise. By default, an OFFSRF stream is in terse mode.

For example, in verbose mode, an **OFFSRF_ofstream** echoes the start and end of all records to **cout**. Thus

```
OFFSRF_ofstream stream("test.out");
stream << verbose
       << bgnrec("TEST RECORD") << 123.45 << endl
              << bgnrec("COMMENT",1) << "This is a comment." << endrec
       << endrec << endl
       << bgnrec("RECORD 2") << " abcDEF" << endrec << endl;
```

will cause the following output on **cout**:

```
{TEST RECORD
  {COMMENT}
}

{RECORD 2}
```

2.7 The skiprecord manipulator

There is an additional manipulator called **skiprecord** which is defined for OFFSRF input streams. It skips over all remaining data and sub-records in the current record. It is especially useful for skipping records which you know will appear in a file but for which your application has no use.

2.8 is_record(), is_data(), and is_eor()

When reading a record which has no fixed amount of data (e.g. a list of names of unspecified length), it is often necessary to know whether the next entry in the file is more data, the start of a sub-record, or the end of the current record. The functions **is_record()**, **is_data()**, and **is_eor()** return this information. Their full prototypes are:

```
int is_record(OFFSRF_ifstream&);
int is_data(OFFSRF_ifstream&);
int is_eor(OFFSRF_ifstream&);
```

The function **is_record()** returns non-zero if the next entry in the file (not including whitespace or comments) is the start of a new sub-record. The function **is_eor()** returns non-zero if the next entry in the file is the end of the current record. The function **is_data()** returns non-zero if the next entry in the file is neither the start of a sub-record nor the end of the current record.

3 The OFFSRF class

The header file **OFFSRF.h** defines a virtual class, **OFFSRF**, which may be used to facilitate the saving and restoring of C++ classes in OFFSRF file records.

The **OFFSRF** class defines three virtual functions with the following prototypes:

```
virtual void read_data(OFFSRF_ifstream &stream);
virtual int read_record(OFFSRF_ifstream &stream, str &name, int skip = 1);
virtual void write(OFFSRF_ofstream &stream);
```

The function **read_data()** is called to read any data at the start of the record which is not contained in sub-records. The base class function **OFFSRF::read_data()** does nothing; therefore, if you are defining a class which reads a record with no data, you need not explicitly include the function **read_data()**.

The function **read_record()** reads a sub-record whose name is **name**; it returns non-zero if the record was read correctly. The base class function **OFFSRF::read_record()** can read **COMMENT** records (see Reference 1, Section 3.1), and **INCLUDE** records (see Section 3.1). The optional argument **skip** is a flag which indicates what **read_record()** should do if it does not recognize the record name in **name**. Usually **skip** is simply passed to **OFFSRF::read_record()** which handles it as follows. If **skip** is non-zero, then if the name is not recognized (i.e. is neither **COMMENT** nor **INCLUDE**), then the **skiprecord** manipulator will be used to skip the record. In this case, the returned value of **OFFSRF::read_record()** will be non-zero. If **skip** is zero, then if the name is not recognized, then **OFFSRF::read_record()** simply returns zero. The main use for the **skip** argument is for the implementation of classes with multiple base classes. An example is given in Section 3.3.

The function **write()** writes all the data for the current record into **stream**. The base class function **OFFSRF::write()** does nothing.

The inserter and extractor operators for OFFSRF streams are overloaded for any class of base type **OFFSRF**. Thus, once the functions **read_data()**, **read_record()**, and **write()** have been defined, an object **obj** of base class **OFFSRF** can be read from an **OFFSRF_ifstream in** by

```
in >> obj;
```

Similarly, it can be written to an **OFFSRF_ofstream out** by

```
out << obj;
```

There is a single constructor for the **OFFSRF** class; the constructor has no arguments. Because the **OFFSRF** class consists solely of virtual functions (with the exception of the constructor), the destructor is declared to be virtual as well; this allows the deletion of an object via a pointer to its **OFFSRF** base class.

3.1 INCLUDE records

The base class function `OFFSRF::read_record()` understands `INCLUDE` records. These records have the form

{`INCLUDE:` *file-name*}

A new `OFFSRF_ifstream` is associated with the file *file-name* and input is read from that file until it is finished. The included file, *file-name*, may include only records, not data; however, data may be included in any sub-records in *file-name*.

The new file is read recursively by calling the virtual function `read_record()` associated with the current `this` pointer. Section 3.4 points out some difficulties that the recursive reading of included files can cause if one is not wary.

3.2 An example

The use of the `OFFSRF` class should become clearer with an example. Suppose we wish to define ship offset data as described in Reference 1, Section 3. For this example only the data describing the transom will be considered.

The transom is represented as a plane which intersects the hull near the stern. The transom plane is defined by specifying a point through which it passes, and a normal vector. These are specified by giving values of four quantities: X, Z, n_X, and n_Z.

The data describing the hull will be read from a file called `ship.in`. The transom is defined by a `TRANSOM` record in this file. The `TRANSOM` record has a sub-record, `TRANSOM OFFSETS` which gives a list of points lying on the edge of the transom. The format of the `TRANSOM` record is

{`TRANSOM:` X Z n_X n_Z
  {`TRANSOM OFFSETS`
    X-value-1 *Y-value-1* *Z-value-1*
    X-value-2 *Y-value-2* *Z-value-2*
    ⋮ ⋮ ⋮
    X-value-n *Y-value-n* *Z-value-n*
  }`TRANSOM OFFSETS`
}`TRANSOM`

One may define the `ship` class to describe complete offset descriptions of a ship, the `transom` class to describe transoms, and the `transom_offset` class to describe a collection of transom offsets. Each of these classes is given a base class `OFFSRF` so that they may be initialized by reading an OFFSRF file.

3.2.1 The transom_offsets class

The `transom_offsets` class implements the offsets as simple arrays along with an integer which gives the number of points.

Since no explicit sub-records are defined for the **TRANSOM OFFSETS** record, there is no need to provide a separate implementation of **read_record()**: the default version provided by the **OFFSRF** base class will do. The default version will read any **COMMENT** records that may be included in the record.

The function **transom_offsets::read_data()** reads the values of the points from the input stream. The function **is_data()** is used to determine whether the next entry in the file is more data. Notice that **is_eor()** should not be used here, since there may be a **COMMENT** sub-record after the offset data. For simplicity, no check is made to ensure that the number of offsets in the file does not exceed **MAX_NO_OFFSETS**.

The function **transom_offsets::write()** simply writes the offset data to the output stream.

```
#include "OFFSRF.h"
#define MAX_NO_OFFSETS 50

class transom_offsets: public OFFSRF
{ int num;
  double X[MAX_NO_OFFSETS], Y[MAX_NO_OFFSETS], Z[MAX_NO_OFFSETS];
public:
  transom_offsets() { num = 0; }

  virtual void read_data(OFFSRF_ifstream &stream)
    { num = 0;
      while (is_data(stream)) stream >> X[num] >> Y[num] >> Z[num++];
    }

  virtual void write(OFFSRF_ofstream &stream)
    { for (int i = 0; i < num; i++)
        stream << X[i] << " " << Y[i] << " " << Z[i] << endl;
    }
};
```

3.2.2 The transom class

The **transom** class contains the four parameters X, Z, n_X, and n_Z, and a pointer to an object of class **transom_offsets**. The function **transom::read_data()** reads the values of X, Z, n_X, and n_Z using the extractor. The pointer value is set by the **transom** class constructor.

The function **transom::read_record()** reads a **TRANSOM OFFSETS** record using the extractor to read the **transom_offsets** object pointed at by **to**. Notice that there is no need to call **transom_offsets::read_data()** explicitly; the extractor handles everything.

If the record name is not understood by **transom::read_record()**, it is passed to the default **read_record()** defined in the base class. This ensures that **COMMENT** and **INCLUDE** records will get read properly.

The function `transom::write()` uses the manipulators `bgnrec` and `endrec` and the inserter `<<` to write the `TRANSOM OFFSETS` record. Again, the function `transom_offsets::write()` need not be called explicitly.

```
class transom: public OFFSRF
{ double X, Z, n_X, n_Z;
  transom_offsets *to;

protected:
  virtual void read_data(OFFSRF_ifstream &stream)
    { stream >> X >> Z >> n_X >> n_Z; }

  virtual int read_record(OFFSRF_ifstream &stream, str &name, int skip=1)
     { int error = 1;
       if (name == "TRANSOM OFFSETS") stream >> *to;
       else error = OFFSRF::read_record(stream,name,skip);
       return error;
     }

  virtual void write(OFFSRF_ofstream &stream)
     { stream << X << " " << Z << " " << n_X << " " << n_Z << endl
              << bgnrec("TRANSOM OFFSETS") << *to << endrec;
     }

public:
  transom(transom_offsets *t): to(t) { }
};
```

3.2.3 The ship class

The `ship` class is defined to read and write the `TRANSOM` record using the extractor and inserter operators. Its only data member is a pointer to a `transom`; the pointer is set when the `ship` is constructed.

```
 class ship:
public OFFSRF { transom *tr;

protected:
  virtual int read_record(OFFSRF_ifstream &stream, str &name, int skip=1)
     { error = 1;
       if (name == "TRANSOM") stream >> *tr;
       else error = OFFSRF::read_record(stream,name,skip);
       return error;
     }
```

```
  virtual void write(OFFSRF_ofstream &stream)
     { stream << bgnrec("TRANSOM") << *tr << endrec << endl;
     }

public:
  ship(transom *t): tr(t) { }
};
```

3.2.4 The function main()

The following example program simply creates objects of class `transom_offsets`, `transom`, and `ship`, then reads data to define them from the file `ship.in`. The data is then written into the file `ship.out`.

```
main()
{ transom_offsets to;
  transom tr(&to);
  ship s(&tr);

  OFFSRF_ifstream in("ship.in");
  in >> s;

  OFFSRF_ofstream out("ship.out");
  out << s;
}
```

Following is a sample input file `ship.in`. The `TRANSOM OFFSETS` record is read from the included file `troffsets.in`.

```
!This is a test file containing ship data
{COMMENT:Reading sample ship data.}

{UNKNOWN RECORD
  {COMMENT:This record should be skipped.}
}

{TRANSOM: 1.0 2.0 3.0 4.0
  {COMMENT:Reading TRANSOM record}
  {INCLUDE: troffsets.in }
}
```

And here is the file `troffsets.in`.

```
!This file contains a TRANSOM OFFSETS record.  It can be included
```

```
!into a TRANSOM record.
{TRANSOM OFFSETS
  1 2 3
  4 5 6
  7 8 9
  {COMMENT:Reading TRANSOM OFFSETS record}
}
```

3.3 Classes with Multiple OFFSRF Base Classes

It is sometimes desirable for a class to be derived from more than one base class, each of which is derived from the OFFSRF class. For example, suppose that class X is derived from the OFFSRF class and knows how to read in the record X-RECORD. Similarly, class Y is derived from the OFFSRF class and knows how to read in the record Y-RECORD.

```
class X: virtual public OFFSRF
{ int x;
protected:
  virtual int read_record(OFFSRF_ifstream &stream, atr &name, int skip=1)
   { int error = 1;
     if (name = "X-RECORD") stream >> x;
     else error = OFFSRF::read_record(stream,name,skip);
     return error;
   }
};

class Y: virtual public OFFSRF
{ int y;
protected:
  virtual int read_record(OFFSRF_ifstream &stream, atr &name, int skip=1)
   { int error = 1;
     if (name = "Y-RECORD") stream >> y;
     else error = OFFSRF::read_record(stream,name,skip);
     return error;
   }
};
```

Now suppose that class XY is derived from both X and Y. (Notice that OFFSRF is a virtual base class for both X and Y since only one copy of the OFFSRF base class is ever necessary.)

```
class XY: public X, public Y
{
protected:
```

```
  virtual int read_record(OFFSRF_ifstream &stream, str &name, int skip=1);
};
```

We wish the `read_record()` function of `XY` to be able to read both `X-RECORD` and `Y-RECORD` records. This can be handled conveniently using the `skip` argument of `read_record()` as follows.

```
int XY::read_record(OFFSRF_ifstream &stream, str &name, int skip)
{ if (X::read_record(stream,name,0)) return 1;
  return Y::read_record(stream,name,skip);
}
```

This function sends the record name to each of its base classes in turn. By setting the `skip` argument to zero when `X::read_record()` is called, we ensure that if it does not understand the name, then it will simply return zero without reading any further in `stream`. Thus, if `X::read_record()` returns zero, then `Y::read_record()` can be called to read the record.

3.4 OFFSRF Classes with OFFSRF Class Members

Suppose that class `X` and class `Y` are both derived from the `OFFSRF` class, but that `X` is a member of `Y`. As in the previous section, we will assume that `X` knows how to read an `X-RECORD` and that `Y` knows how to read a `Y-RECORD`.

```
class X: virtual public OFFSRF
{ int x;
public:
  virtual int read_record(OFFSRF_ifstream &stream, atr &name, int skip=1);
};

class Y: virtual public OFFSRF
{ int y;
  X xtemp;
protected:
  virtual int read_record(OFFSRF_ifstream &stream, atr &name, int skip=1);
};
```

Commonly, the OFFSRF data file is set up to reflect the class hierarchy, so that the `X-RECORD` will be a sub-record of the `Y-RECORD`:

```
{Y-RECORD: 123.4
  {X-RECORD: 567.8 }
}Y-RECORD
```

In this case, the function `Y::read_record()` is straightforward.

```
int Y::read_record(OFFSRF_ifstream &stream, atr &name, int skip=1);
{ int error = 1;
  if (name = "X-RECORD") stream >> xtemp;
  else error = OFFSRF::read_record(stream,name,skip);
  return error;
}
void Y::read_data(OFFSRF_ifstream &stream)
{ stream >> y;
}
```

However, suppose that it is desired that X-RECORD and Y-RECORD have an equal footing in the OFFSRF data file:

```
{X-RECORD: 567.8 }
{Y-RECORD: 123.4 }
```

When the X-RECORD is read we want its value to initialize Y::xtemp.x. This can be achieved using the skip argument to Y::read_record().

```
int Y::read_record(OFFSRF_ifstream &stream, atr &name, int skip=1);
{ error = 1;
  if (str == "Y-RECORD") stream >> y;
  else error = xtemp.read_record(stream,name,skip);
  return error;
}
```

Notice that the class Y must have access to the X member function X::read_record(); hence, either X::read_record() should be declared public (as above), or Y should be declared a friend of the class X.

The above example of Y::read_record() has a not-very-obvious problem. Suppose that instead of an X-RECORD or Y-RECORD, there is an INCLUDE record which directs input to a file where the X-RECORD and Y-RECORD are found. The record Y-RECORD will not be read from the included file. Why not? When Y::read_record() encounters the INCLUDE record, control is passed via X::read_record() to OFFSRF::read_record(), which opens a new input stream and starts to read the records within it recursively. To read the records, the function X::read_record() is used, since the current this pointer points to an object of class X. Since X:read_record() does not know how to read a Y-RECORD record, it will be skipped.

The solution to the problem is to call OFFSRF::read_record() *before* X::read_record() to pick up any INCLUDE records.

```
int Y::read_record(OFFSRF_ifstream &stream, atr &name, int skip=1);
{ error = 1;
  if (str == "Y-RECORD") stream >> y;
  else if (OFFSRF::read_record(stream,name,0))  return 1;
  else error = xtemp.read_record(stream,name,skip);
  return error;
}
```

4 Concluding Remarks

A set of classes to ease the use of OFFSRF data files has been described. The parsing of the OFFSRF record structure is handled by defining a input stream class **OFFSRF_ifstream**. The associated output stream class **OFFSRF_ofstream**, along with its manipulators **bgnrec** and **endrec**, provide a simple means of writing output files which are properly indented and which have matching begin and end record markers.

Several manipulators have been defined for these streams in order to provide the programmer with some flexibility in their use. Of course, all the standard **istream** and **ostream** manipulators will also work.

Classes can easily be made to read OFFSRF records by deriving them from the **OFFSRF** base class and defining the three virtual functions **read_record()**, **read_data()**, and **write()**. No parsing of the OFFSRF record names is necessary. Moreover, the insertion (**>>** and extraction (**>>**) operators have been overloaded for all classes derived from the **OFFSRF** base class, so that reading the file can be done in a very concise and elegant way.

Appendix

A The str class

The class **str** is a representation of character strings. It is safer to use than ordinary arrays of **char** since memory usage is strictly controlled. In addition, there are many member functions which make the **str** class easy to use. In the **OFFSRF** class uses the **str** class to represent record names.

Automatic type conversion from **str** to **char*** is provided so that a **str** can be used as an argument to any function which requires a **char***; however, type conversion from **char*** to **str** must be done explicitly using either the string constructors or the function **ch2str()** which, given a **const char*** argument, returns the equivalent **str**.

The definition of the **str** class is contained in the header file **str.h**.

A.1 Constructors

An instance of a **str** may be declared in five ways.

str s: declares **s** to be a string of length zero. No memory is allocated for the characters in the string.

str s(n): where **n** is an integer, declares **s** to be a string of length **n**. Enough memory will be allocated for **n+1** characters; the extra memory location is so that a trailing null character can be appended if the **str** is converted to **char***.

str s(n,m): where **n** and **m** are integers, declares **s** to be a string of length **n**. Enough memory is allocated for either **m** or **n** characters, whichever is greater.

str s(t): where **t** is another **str** initializes **s** by copying **t**.

str s(t): where **t** is of type **char*** initializes **s** by copying **t** up to the first null character. Enough memory is allocated to contain the all characters including the null.

A.2 Member Functions

The following member functions are defined for the **str** class.

int len(): returns the length of the string.

int max_len(): returns the maximum length of the string available with the current memory allocation.

void set_len(int n): changes the length of the string to **n**. If there is not enough memory currently allocated, more will be allocated and the current contents of the string will be copied to the new memory.

void set_mem(int m): changes the memory allocation so that a string of length **m** can be stored. If **m** is less than the current allocation, nothing is done.

void strip_leading_ws(): strips whitespace from the beginning of a string (whitespace is defined as any characters for which the C function **isspace()** returns true).

void strip_trailing_ws(): strips whitespace from the end of a string.

void strip_ws(): strips whitespace from the beginning and the end of a string.

void shiftr(int n, int lo, int hi): shifts the characters between **lo** and **hi**, **n** places to the right. If a character is shifted past the end of the string, it is lost. If **n** is negative, the shift is to the left.

void shiftl(int n, int lo, int hi): is like **shiftr()** but shifts the characters to the left for positive **n** and to the right for negative **n**.

void shiftc(int n): does a circular shift of the whole string by **n** places to the right: i.e. the array is shifted to the right but characters which are shifted past the end of the array are copied to the beginning. If **n** is negative, the shift is to the left.

void insert(const char &t, int i): inserts the character **t** at the i^{th} position in the string. The length of the string is increased by one.

void insert(const str &s, int i): inserts the string **s** into the current string starting at the the i^{th} character. The length of the string is increased by the length of **a**.

ifstream &read(ifstream &stream): reads a **str** from a binary input stream. The length, **n**, is read first, then the next **n** bytes in the file are read and interpreted as characters.

ofstream &write(ofstream &stream): writes a **str** to a binary output stream.

A.3 Overloaded Operators

The following operators are overloaded for the **str** class. Let **s1** have type **str** and let **s2** have type **str** or **char***.

s1[i]: returns element **i** of **s1**.

s1 = s2: copies **str s2** to **s1** and returns **s1**.

s1(i,j): returns a *copy* of the sub-string from element **i** to element **j**.

s1 += s2: appends **s2** to **s1** and returns the value.

s1 + s2: returns a **str** created by concatenating **s1** and **s2**.

s1 == s2: returns true (non-zero) if the length of **s1** and **s2** are the same and the characters in each are the same.

s1 != s2: returns **!(s1 == s2)**.

A.4 Inserter and Extractor

The inserter **<<** writes a **str** to an **ostream**. The extractor **>>** reads characters from an **istream** until whitespace is found. These functions are equivalent to reading/writing a **char*** before/after conversion to **str**.

Index

References

1. D. Hally, "OFFSRF: A System for Representing Ship Offset Data," DREA Technical Communication 89/305, 1989.

2. D. Hally, "User's Guide for HLLFLO Version 2.0," DREA Technical Communication 93/309, 1993.

DOCUMENT CONTROL DATA

(Security classification of title, body of abstract and indexing annotation must be entered when the overall document is classified)

1. ORIGINATOR (The name and address of the organization preparing the document. Organizations for whom the document was prepared, e.g. Establishment sponsoring a contractor's report, or tasking agency, are entered in section 8.)

Defence Research Establishment Atlantic
P.O. Box 1012, Dartmouth, N.S. B2Y 3Z7

2. SECURITY CLASSIFICATION (Overall security of the document including special warning terms if applicable.)

Unclassified

3. TITLE (The complete document title as indicated on the title page. Its classification should be indicated by the appropriate abbreviation (S,C,R or U) in parentheses after the title.)

C++ Classes for Reading and Writing Files in OFFSRF Format

4. AUTHORS (Last name, first name, middle initial. If military, show rank, e.g. Doe, Maj. John E.)

Hally, David

5. DATE OF PUBLICATION (Month and year of publication of document.)

February 1994

6a. NO. OF PAGES (Total containing information. Include Annexes, Appendices, etc.)

25

6b. NO. OF REFS. (Total cited in document.)

2

6. DESCRIPTIVE NOTES (The category of the document, e.g. technical report, technical note or memorandum. If appropriate, enter the type of report, e.g. interim, progress, summary, annual or final. Give the inclusive dates when a specific reporting period is covered.)

DREA Technical Communication

8. SPONSORING ACTIVITY (The name of the department project office or laboratory sponsoring the research and development. Include the address.)

Defence Research Establishment Atlantic
P.O. Box 1012, Dartmouth, N.S. B2Y 3Z7

9a. PROJECT OR GRANT NUMBER (If appropriate, the applicable research and development project or grant number under which the document was written. Please specify whether project or grant.)

1AN

9b. CONTRACT NUMBER (If appropriate, the applicable number under which the document was written.)

10a. ORIGINATOR'S DOCUMENT NUMBER (The official document number by which the document is identified by the originating activity. This number must be unique to this document.)

DREA Technical Communication 94/302

10b. OTHER DOCUMENT NUMBERS (Any other numbers which may be assigned this document either by the originator or by the sponsor.)

11. DOCUMENT AVAILABILITY (Any limitations on further dissemination of the document, other than those imposed by security classification)

(X) Unlimited distribution
( ) Distribution limited to defence departments and defence contractors; further distribution only as approved
( ) Distribution limited to defence departments and Canadian defence contractors; further distribution only as approved
( ) Distribution limited to government departments and agencies; futher distribution only as approved
( ) Distribution limited to defence departments; further distribution only as approved
( ) Other (please specify):

12. DOCUMENT ANNOUNCEMENT (Any limitation to the bibliographic annoucement of this document. This will normally correspond to the Document Availability (11). However, where futher distribution (beyond the audience specified in 11) is possible, a wider announcement audience may be selected.)

13. ABSTRACT (A brief and factual summary of the document. It may also appear elsewhere in the body of the document itself. It is highly desirable that the abstract of classified documents be unclassified. Each paragraph of the abstract shall begin with an indication of the security classification of the information in the paragraph (unless the document itself is unclassified) represented as (S), (C), (R), or (U). It is not necessary to include here abstracts in both offical languages unless the text is bilingual)

The OFFSRF format defines a record structure for files to be used as input to computer programs. The original application was offset data for ship hulls, but the format itself is very general. It is most suitable for comparatively small sets of data which are often modifed using ordinary text editors.

When OFFSRF was originally designed, several FORTRAN subroutines were provided to ease the use of the format for programmers. In this document the programmer support is extended to the language C++. C++ classes are described which can be used to read and write files in OFFSRF format.

14. KEYWORDS, DESCRIPTORS or IDENTIFIERS (Technically meaningful terms or short phrases that characterize a document and could be helpful in cataloguing the document. They should be selected so that no security classification is required. Identifiers, such as equipment model designation, trade name, military project code name, geographic location may also be included. If possible keywords should be selected from a published thesaurus. e.g. Thesaurus of Engineering and Scientific Terms (TEST) and that thesaurus-identified. If it is not possible to select indexing terms which are Unclassified, the classification of each should be indicated as with the title.)

computer programs
C++
OFFSRF